Part III

The Monitoring Disciplines

Chapter 7

Metrics, Cost, and Estimation

Not everything that can be counted counts, and not everything that counts can be counted — Albert Einstein.

In Part III of these notes, we will discuss one of the primary responsibilities of the project management team: *monitoring* the execution of the project.

Monitoring typically consists of measuring and analysing information about the project, such as:

1. providing estimates on the future performance of the project;
2. comparing these estimates to the actual performance; and
3. assessing the quality of the outputs of various activities of the project.

This chapter, Chapter 7, discusses metrics for measuring and estimating complexity, cost, and effort of a project, so that effective planning can be undertaken. Chapter 8 is dedicated to risk management: the assessment and control of events that pose risks to the project. Chapter 9 discusses quality assurance, and some of the approaches used to monitor the quality of project outputs.

7.1 Metrics

Why measure software?

The first question that needs to be answered is: why do we want to measure software at all? Software is functional, so do we care how big it is as long as it works?

Florac and Carleton [FC99] identify four key reasons that measuring software is useful.

1. **To Characterise** — to gain understanding of the process, products, resources and environments, and to establish baselines for comparison with future assessments.
2. **To Evaluate** — to determine status with respect to plans.
3. **To Predict** — to gain an understanding of the relationships among processes and products, and use this understanding to build models of these relationships for the purpose of estimating future performance.
4. **To Improve** — to identify roadblocks, root causes and other opportunities for improving product quality and process performance.

Using metrics wisely

Software projects rely heavily on people, so metrics need to be used wisely. They can be a destructive force just as easily as they can be a good tool to help you improve your software, so we must take care not to use them unwisely. Some good guidelines for applying metrics due to Grady [Gra92] are:

- Provide regular feedback to the individuals and teams who collect measures and metrics.
- Work with engineers and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.
- Metrics data that indicate problems should not be considered "negative". All that metrics do is suggest areas for further scrutiny.
- Do not get obsessed with a single metric — for example, defect rates — to the exclusion of all other metrics.
- Use common sense and organisational sensitivity when interpreting data.

Types of metrics

In a software engineering project, there are three types of metrics that can be employed.

1. **Process Metrics** — These metrics are collected across all projects over long periods of time for long term *process improvement*.
2. **Project Metrics** — These are metrics that are collected over a single project to:
   - assess the status of ongoing projects;
   - track potential risks;
   - uncover problem areas before they become critical;
   - adjust workflow; and
   - evaluate a team's ability to control quality.
3. **Product Metrics** — These metrics measure the attributes of a product as it evolves. They are collected for assessing the *quality* of a product.

It is common to employ all of these on a project. Ultimately, it is the product metrics that matter, because it is the product that we are selling. However, product metrics are not direct measures of software quality, like measures in other engineering disciplines; e.g. voltage, velocity, or temperature. More importantly, it is not possible to measure product quality unless there is a product to measure, Therefore, product measurement is not possible until late in the development lifecycle — a point at which it is too late to change the quality of the product.

Instead, software engineers also measure process and project metrics. The reasoning behind this is that, if we cannot directly measure the product quality, then we can instead measure aspects of the processes that are used to create the product, and measure aspects of the project — the results of applying the processes.

Figure 7.1 presents an overview of the processes involved in process, project, and product measurement. First, data must be collected on the item being measured. Then, one must calculate the summary information using a metric calculation. Finally, indicators are used to interpret the metric in a context.

In this subject, our focus will be on metrics for *estimation* — that is, how to estimate the size and complexity of a software project for the purpose of planning — because these are primarily the metrics used by a project manager.

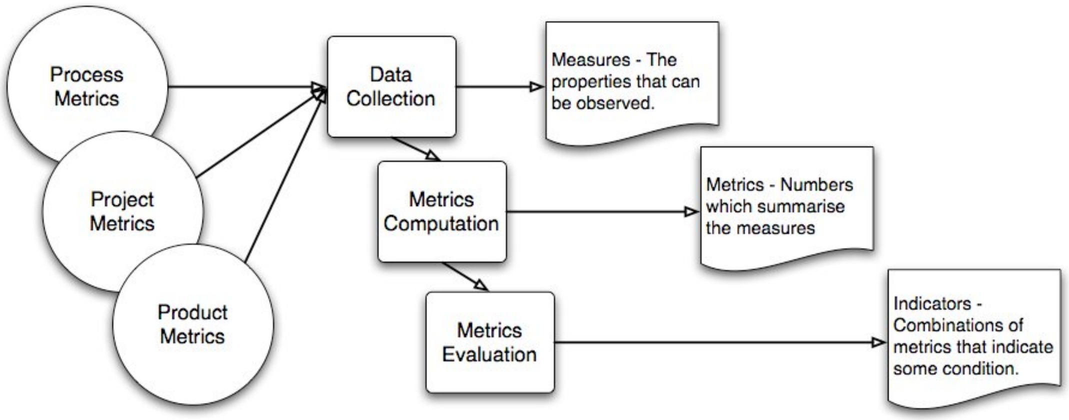

Process Metrics
Project Metrics
Product Metrics
Data Collection
Metrics Computation
Metrics Evaluation
Measures - The properties that can be observed.
Metrics - Numbers which summarise the measures
Indicators - Combinations of metrics that indicate some condition.

Figure 7.1: An overview of the processes involved in process, project, and product measurement.

Attributes of useful metrics

Ejiogu [Eji91] proposes the following six attributes that make a metric useful:

1. **Simple and computable** — calculating (and learning how to calculate a metric) should be straightforward.
2. **Empirically and intuitively persuasive** — a metric should appear valid when proposed, and be in line with the expectations and experience of software engineers.
3. **Consistent and objective** — two different people applying the same metric to the same artifact should arrive at the same answer.
4. **Use of consistent units** — the computation should not lead to combinations of units; e.g. multiplying people on the project by lines of code is not intuitively persuasive.
5. **Programming language independent** — metrics should not be based on the syntax of programming languages, as these differ so much that different languages will end up with wildly different results.
6. **Useful for providing feedback** — a metric should be useful for providing feedback to improve the artifact.

These attributes are meant only as a guideline for effective metric design. Some metrics do not satisfy all of the above, yet many of these are still useful. However, it seems unlikely that a metric satisfy very few of these would be useful or effective.

Example 7.1. *Source Lines of Code (LOC)*

A common (and easy to understand) example of a product metric is the number of lines of source code (LOC). By measuring the lines of code in a product, we get a measure of the effort that was expended to produce that product, and the effort required to understand that product. While this is not entirely accurate — that is, a project with more lines of code does not always take more effort — it is a good approximation. It would be hard to argue that a product with one thousand lines of code took as much effort as one with one million lines of code.

There are several different ways to calculate LOC, which can be broadly separated into the two categories: *physical* lines of code, or *logical* lines of code. Physical metrics count the number of lines including comments, and sometimes blank lines. Logical metrics count the number of logical statements in the programming, ignoring comments and blank lines. These must be tied to particular programming languages, because the shape and style of statements differ. In C, for example, a common approach is to count the number of statement-terminating semi-colons.

Relating this metric to the processes in Figure 7.1, the data collection is the source code itself. The method of metric calculation is to count the number of lines of code, as discussed in the previous paragraph; e.g. logical LOC.

Finally, an indicator could categorise the LOC measurement on the basis of size to give an measure of the difficulty of maintenance. For example:

Indicator	Meaning
LOC $\leq$ 1000	straightforward
1000 $<$ LOC $<$ 100,000	medium
LOC $\geq$ 100,000	difficult

7.2 Metrics for measuring complexity

In this section, we present several metrics for measuring complexity in software. Specifically, we present:

Cyclomatic complexity — a measure of complexity for source code that determines the number of different paths in a program.

The CK metrics suite – a suite of complexity measures for object-oriented designs.

Function point analysis — a measure of complexity for requirements models that estimates the amount of functionality in a system.

7.2.1 Cyclomatic complexity for measuring program complexity

Counting the number of lines of code is one way to measure the complexity of the code in a software application. However, as discussed in Example 7.1, it is dependent on the underlying programming language, and whether or not certain lines are counted. For example, one can choose to not count comments, however, is commenting not part of programming?

Other measures of code complexity attempt to measure the complexity with respect to how many decision points exists in a program. This is deemed to be a better complexity measure that is far less dependent on the programming language. Cyclomatic complexity, proposed by McCabe [McC76], is the most common of such methods.

Definition 7.1. *Cyclomatic complexity*
The cyclomatic complexity of a program is the upper bound of the number of *linearly independent paths* in the code of that program. A set of linearly independent paths is a set of paths through the program, in which each path executes at least one statement/branch that is not part of any other path in the set.

Calculating cyclomatic complexity

To calculate the cyclomatic complexity of a graph, one first converts the program into its *control-flow graph*. A control-flow graph is a directed graph representing all paths that can be executed by a a computer program. The graph, consisting of nodes that represent statements and branches, and direct edges that represent flow between those nodes — that is, an edge exists between two nodes is the statements/branches represented by those nodes can be executed one after the other.

As an example of a control-flow graph, consider the program in Figure 7.2 for calculating the greatest common divisor of two integers using Euclid's algorithm, and its corresponding control-flow graph in the same figure. The control-flow graph contains two types of nodes: boxes, which represent statements (in fact, blocks of statements), and diamonds, which represent branches (or choices).

From graph theory, we know that the maximum number of linearly independent paths in a graph is equal to

$$e - n + 2$$

in which e is the number of edges and n the number of nodes. Thus, for the control-flow graph in Figure 7.2, there are five nodes and six edges, so the cyclomatic complexity is $6 - 5 + 2 = 3$.

McCabe [McC76] proved an equivalent (a perhaps more straightforward) method for calculating cyclomatic complexity of a control-flow graph: count the number of branches/decisions (diamonds in the control-flow graph) and add

```
int gcd(int x, int y)
{
  while (x != y) {

    if (x > y) {
      x = x - y;
    }
    else {
      y = y - x;
    }
  }

  return x;
}
```

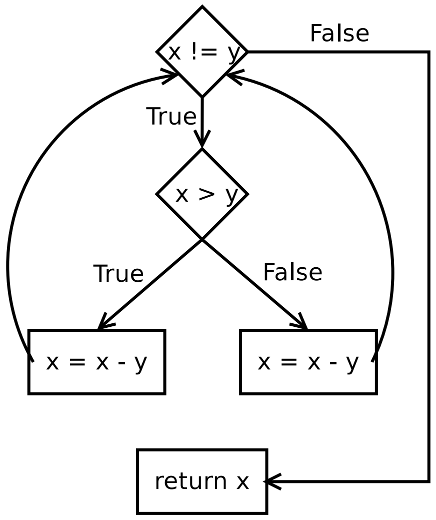


Figure 7.2: An example program for calculating the greatest common denominator of two integers using Euclid's algorithm, and its control-flow graph.

one. That is, McCabe proved that the upper bound of the number of linearly dependent paths in a program is equivalent to $D+1$, where D is the number of decision points.

Applications of cyclomatic complexity

There are typically three applications of cyclomatic complexity.

Measuring complexity of a program — Cyclomatic complexity is an estimate of how much a developer has to track and examine when attempting to understand a program. McCabe's original motivation for developing the measure was for developers to measure the complexity of the code they were writing, and to re-factor the code into multiple procedures or modules if the complexity was too high.

Testing — Cyclomatic complexity gives us some idea of the amount of test cases we will need to generate to achieve certain coverage measures. If the cyclomatic complexity of a program is C, then C is the most number of test cases required to execute every branch/decision in the program. Additionally, C is the least number of test cases required to execute every path in the program. Watson, McCabe, and Wallace [WMW96] discuss a testing technique that generates a test case for every linearly dependent path in a program.

Defect estimation — As state above, cyclomatic complexity is an estimate of how much a developer has to track and examine when attempting to understand a program. This also applies to the original programmer. A program with a higher cyclomatic complexity is likely to be more complex to develop. As such, researchers have proposed that a program with higher cyclomatic complexity is more likely to contain more faults. A recent study (available at `http://www.enerjy.com/blog/?p=198`) showed a strong correlation between cyclomatic complexity a fault count in Java classes. Classes with a complexity of 11 had a 28% chance of being fault-prone, while the most complex classes with a complexity of 74 had a 98% chance of being fault-prone. This indicates that, when testing a software system, we may want to spend more effort testing the modules with a higher cyclomatic complexity. It can also be used to estimate the reliability of a program without running tests.

7.2.2 The CK metrics suite for measuing object-oriented design complexity

The CK metrics suite was proposed by Chidamber and Kemerer [CKM94] as a measure of the complexity of an object-oriented design. The suite consists of six different measures of an object-oriented design.

The metrics are based around classes — the fundametnal unit of an object-oriented system. By measuring how large classes are, how the methods within a class relate and interact, and how classes interact with each other, we can get an idea of the complexity of the system.

Chidamber and Kemerer propose the following six metrics.

Weighted methods per class (WMC) — This metric requires a measure of the complexity of each method in a class interface, such as the cyclomatic complexity in Section 7.2.1. If the source code or internal design are not available, this must be estimated.

If a class interface has n number of methods, with complexity measures $c_1, c_2, \ldots, c_n$, then the WMC of that class is

$$\sum_{i=1}^{n} c_i.$$

That is, a straight summation of the complexity of all methods in the class.

Chidamber and Kemerer recommend minimising the value of WMC for a class. First, the higher the WMC, the more likely it is that the class is specific to the application, thus limiting its reuse. Second, the higher the WMC, the more complex the inheritance tree below this class becomes, because all classes inherit the methods.

Depth of inheritance tree (DIT) — This is simply the maximum length path from the root class to a leaf node (class without any subclasses) in an inheritance tree. Figure 7.3 shows an example inheritance hierarchy with a DIT of four: from the root class C to the leaf class C2.1.1.

A higher DIT can be considered both positive and negative. On one hand, it implies high reuse (via inheritance), which is good, but on the other hand, the classes lower down the hierarchy may contain a lot of methods (hence a higher WMC), which makes the behaviour of the class difficult to understand, especially if there is overriding and overloading of methods. Also, a deep inheritance hierarchy indicates a higher design complexity.

Number of children (NOC) — This is a straightforward count of the number of *direct* children of a class. In Figure 7.3, we have the following NOC measures:

Class	NOC
C	2
C1	1
C2	3
C2.1	1

with the remaining class each having a NOC measure of 0.

As with DIT, a high NOC value can be both positive and negative. Again, a high NOC indicates reuse, but it may also be that the abstraction that a parent class represents may be diluted to the point that instances of the children class do not fulfil the specified behaviour for the parent class.

Response for a class (RFC) — The response set for a method in a class is the number of method calls made by that method if it is invoked. The RFC for a class is the summation of the size of the response set for all methods in a class. Given a class with n methods and response sets $r_1, r_2, \ldots, r_n$ for each method, the RFC of the class is

$$\sum_{i=1}^{n} \#r_i.$$

A higher RFC value for a class indicates a higher complexity, due to the increase in communication/interaction with other classes and methods in the system.

As with the WMC metric, this metric requries us to either have access to the implementation of each method, or to estimate the number of method calls that will be used.

Coupling between object classes (CBO) — The CBO for a class is the number of relationships that it has with other classes, other than via inheritance, such as aggregation and association. A class A is coupled with a class B if either of them "act upon" each other. Coupling is bi-directional — that is, if class A is coupled to class B, then class B is coupled to class A.

A class with a higher CBO is less likely to be reusable than one with a low CBO due to the dependencies involved. In addition, high value CBOs make maintenance and testing of classes more difficult, due to the number of interactions that occur between instances of the different classes.

Lack of cohesion in methods (LCOM) — Each method in class accesses (reads or writes to) zero or more of the attributes (e.g. instance variables) in that class. The LCOM value of a class if the number of pairs of methods whose similarity is zero, minus the number of pairs of methods who similarity is not zero. Similarity is defined as the number of attributes that are accessed by both methods.

For example, if we have three methods, m_1, m_2, and m_3, in a class, and they access the following attributes:

$$\begin{aligned} m_1 &= \{v_1, v_2\} \\ m_2 &= \{v_2, v_3\} \\ m_3 &= \{v_4\}. \end{aligned}$$

We enumerate all possible Cartesian pairs of methods in the class, and calculate the attributes they share:

$$\begin{aligned} m_1 \cap m_2 &= \{v_2\} \\ m_1 \cap m_3 &= \{\} \\ m_2 \cap m_3 &= \{\}. \end{aligned}$$

Two pairs have a similarity of zero, while the other has a non-zero similarity. Therefore, the cohesion of the method is $2 - 1 = 1$.

The "lack of" in the name of this metric is to keep consistency with the other metrics, in that a higher value is "bad". Unlike coupling, a good design should have a high cohesion; that is, methods in a class should be accessing the same attributes as each other, otherwise they are unrelated. If a method accesses only one variable, and that variable is only accessed by that method (e.g. method m_3 and variable v_4 above), then the variable and method should perhaps be refactored into another class.

This is somewhat confusing at first, but one must remember that high cohesion is considered good, but that a high LCOM value is not, as it represents low cohesion. If all methods are similar (access the same attribute), then LCOM is 0.

It is recommended that each of the values in this metric suite be kept as low as reasonable. A study by Mishra and Bhavsar [MB03] of 30 systems implemented in C++ indicate that an increase in any one of the values for the above metrics increases the density of faults and decreases the quality of the software.

The coupling and cohesion metrics are applicable to module- and component-based systems, and the same rationale for the metrics apply. These two metrics together are generally used to measure the quality of software designs, and are related to each other: low coupling often correlates with high cohesion, and vice versa.

7.2.3 Function point analysis for measuing requirements complexity

Perhaps the most widely used metric for measuring the complexity of requirements models is *function points*.

Definition 7.2. *Function point*
A function point is a unit of measurement that is used to express the amount of functionality in a software system, as seen by the user.

A higher number of function points indicates more functionality, and empirical evidence demonstrates that there is a positive correlation between function points and the complexity of the system. Function points are typically used to:

- estimate the cost and effort required to design, code and test a software system;

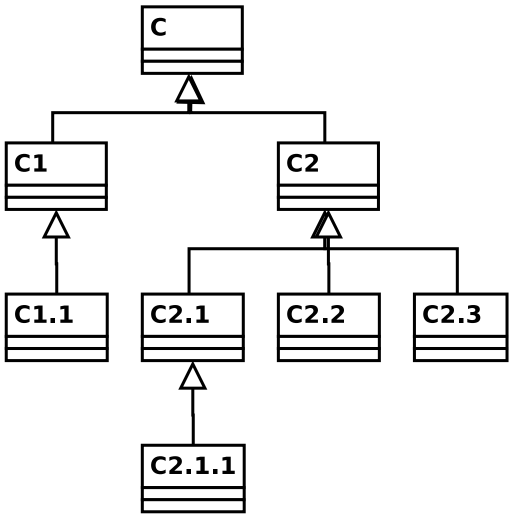


Figure 7.3: A class hierarchy

- predict the number of errors; and
- predict the number of components in the software.

Calculting the number of function points in a system is a five step process:

1. From the user requirements, categories each functional requirement into one of five different categories.
2. Weight the complexity of these categories for the particular application.
3. Using a template, calculate the *count total* from the categories and their complexity.
4. Calculate the *value adjustment factors*, which weight the non-functional requirements into the estimate.
5. Using a formula, calculate the total function points count.

This sections details the above five steps, and presents an example of function point analysis.

Step 1 — Categorising functions

The first step in calculating the number of function points to collect data from the requirements specification. Given a set of functional user requirements, one must categorise each of these requirements into one of the following.

- **Internal logical file (ILF)** — a logical grouping of data that the system maintains over a period of time, and is modified using external inputs.

  Examples of ILFs include tables in a relational database, files containing user settings, or data structures holding information about the system state.

- **External interface file (EIF)** — a logical grouping of data that is maintained external to the system, but which may be used by the system.

  Examples of EIFs are the same as ILFs, except that the data is maintained outside the system, such as data hosted on a third-party server.

- **External input (EI)** — an input to the system from a user or another application, which is used to control the flow of the system, or provide data. External inputs generally modify internal logic files.

  Examples of EIs include data fields populated by users, inputs files (e.g. program source code to a compiler), and file feeds from an external application.

- **External outputs (EO)** — an output to the user that provides information about the state of the system.

  Examples of EOs include screens, error messages, and reports that are shown to the user. Individual data fields in these are grouped as one external output.

- **External inquiries (EQ)** — an online input for which the system responds immediately via an online output. The input is not used to update an internal logic file, but is used to query the internal logic file and provide an output. The output is retrieved directly, with no derived data included.

  Examples of EQs are similar to that of EOs, except in cases in which the output does not include any derived data from the system, such as reading a user setting, or reading a record from a database table.

The first two of these functions are *data* functions — that is, they are concerned only with the maintenance of data for the application — while the remaining three are *transaction* functions — that is, they are concerned with information being passed to and from the system.

Note that some functions can fall into more than one category. For example, a third-party system from which our system reads information could be either an EIF or an EI, depending on the use of the data. A rule of thumb is that is the system sends data *back* to the external system, then it is an EIF, otherwise, it is only input, so is an EI.

Finally, *count* the number of each category; that is, count how many ILFs there are, etc.

Step 2 — Assigning complexity values

Once each functional unit is classified, a *complexity value* is associated with each category. The complexity of a category is ranked either *simple*, *average*, or *complex*.

The complexity value is quite crude: it represents the complexity of all functions in a category — for example, all external inputs —, rather than the complexity of each user requirement. However, applying the complexity to each function would be straightforward.

A technique for estimating the complexity value is to count the number of *data element types* (DETs), *record element types* (RETs), and *file type references* (FTRs).

A DET is a *unique*, user-visible data field in a system, including inputs, outputs, and internal files. For example, an internal file storing information about a customer may contain an email address, which is unique to each customer.

A RET is a user-visible *subgroup* of data fields in an ILF or EIF. That is, data files that are of the same type, but belong to a single instance. For example, an internal file may contain a unique customer email address, however, that customer may have several delivery addresses, which are grouped together as a RET. As a rule of thumb, and RET is generally related to a DET as a "child" — a one-to-many relationship between a DET and an RET.

A FTR is a file referenced by a transaction. An FTR must be an ILF and EIF. For this, we count the number of FTRs for each transaction function (EI, EO, and EQ).

The relationship between DETs, RETs, FTRs, and the function categories, is summarised by the following table:

Function	**DETs**	**RETs**	**FTRs**
Internal Logical Files (ILFs)	✓	✓	
External Interface Files (EIFs)	✓	✓	
External Inputs (EIs)	✓		✓
External Outputs (EOs)	✓		✓
External Inquiries (EQs)	✓		✓

Once each of these is counted, calculating the complexity value is a straightforward lookup from Table 7.1. Note that the calculation depends on what type of function we are measuring: either a data function (ILFs and EIFs) or a transaction function (EIs, EOs, and EQs).

(a) Complexity value table for data functions

	DETs		
RETs	1-19	20-50	51+
1	Simple	Simple	Average
2-5	Simple	Average	Complex
6+	Average	Complex	Complex

(b) Complexity value table for transaction functions

	DETs		
FTRs	1-5	6-19	20+
1	Simple	Simple	Average
2-3	Simple	Average	Complex
4+	Average	Complex	Complex

Table 7.1: Tables for calculating complexity values from DET, RET, and FTR count.

Step 3 — Calculating the count total

Using the count and complex estimate, the template table in Figure 7.2 is completed, and the *count total* is calculated. The weighting factors for the different complexity values have been calculated by analysing data from existing projects.

Information Domain Value	**Count**		**Weighting Factor Simple**	**Average**	**Complex**		
Internal Logical Files (ILFs)		×	7	10	15	=	
External Interface Files (EIFs)		×	5	7	10	=	
External Inputs (EIs)		×	3	4	6	=	
External Outputs (EOs)		×	4	5	7	=	
External Inquiries (EQs)		×	3	4	6	=	
Count total							

Table 7.2: A template table for calculating the number of function points in a system.

As discussed in the previous step, one could define the complexity value for each function, rather than each category. For this, one would obtain the count total by simply summing the weighting factors of each function in the requirements model, instead of applying the template in Table 7.2.

Step 4 — Calculating the value adjustment factors

The next step is to provide *value adjustment factors* (VAF). These factor in the non-functional characteristics of the system, and some detail relevant to the design architecture. The VAFs are based on the answers to the following fourteen questions:

1. Does the system require reliable backup and recovery?
2. Are specialised data communications required to transfer information to or from the application?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system execute in an existing heavily utilised operational environment?
6. Does the system require interactive/on-line data entry?
7. Does the online data entry require the input transaction to be built up over several screens?
8. Are the Internal Logical Files updated online?

9. Are the inputs, outputs, files or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system to be designed for multiple installations in different organisations?
14. Is the application designed to facilitate change and for ease of use by the user?

The answers for each of these is on a scale of 0 to 5, with 0 meaning that the factor is not important, and 5 meaning that it is critical.

Step 5 — Calculating the function point count

In the final step, the count total and value adjustment factors are plugged-in to the following formula to estimate the function point count:

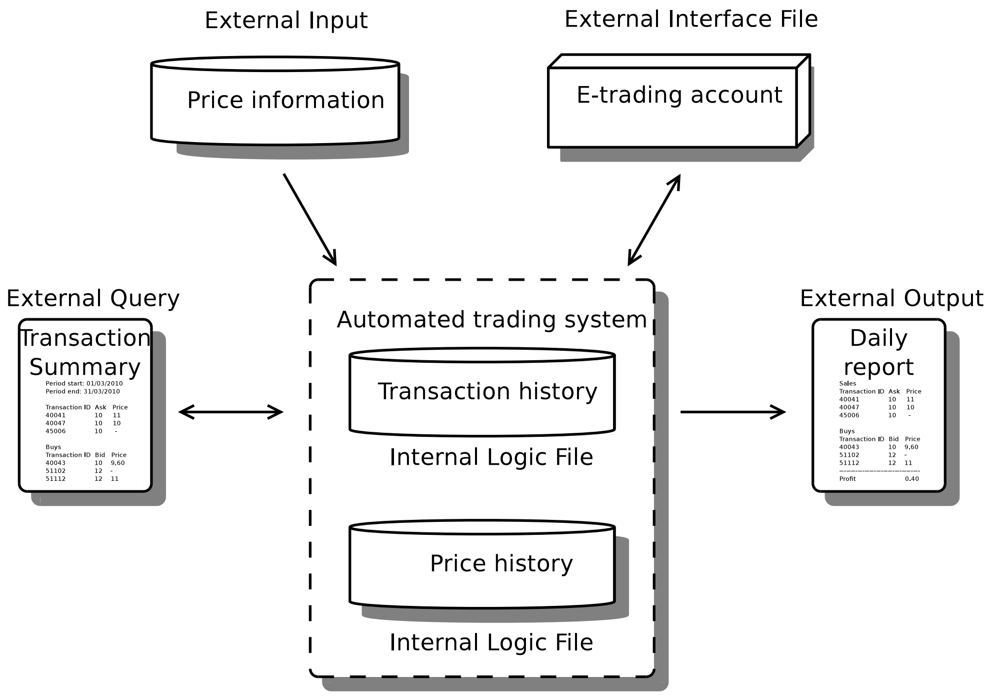
$$\mathrm{FP} = \text{count total} \times (0.65 + 0.01 \times \sum_{i=1}^{14} F_i),$$

in which F_i is the VAF corresponding to i-th VAF question.

Therefore, the number of function points is the count total, multiplied by a scaling factor, which starts at 0.65, and increases by 0.01 for every point in every VAF rating.

Example 7.2. *Calculating function points for an automated trading system*

Consider a team writing an application that automatically trades on a commodities market using a new trend analysis strategy. Figure 7.4 shows a high-level architectural overview of the system.



```
Period start: 01/03/2010
Period end: 31/03/2010

Transaction ID  Ask   Price
40041           10     11
40047           10     10
45006           10      -

Buys
Transaction ID  Bid   Price
40043           10    9.60
51102           12    -
51112           12    11
```

```
Sales
Transaction ID  Ask   Price
40041           10     11
40047           10     10
45006           10      -

Buys
Transaction ID  Bid   Price
40043           10    9.60
51102           12    -
51112           12    11
-------------------------
Profit                0.40
```

Figure 7.4: A high-level architectural view of the automated trading system.

Each trading day, the system reads in yesterday's trading information from a third-party server (external input): the high price, low price, opening price, and closing price. A complete history is saved in a "price history" database (internal logic file). Based on the trends in prices for particular commodities, the system will decide which commodities to bid for, and which to try to sell. This information is sent to an external e-trading account (external interface file), which places the bid/ask for each commodity. When a bid/ask is either accepted or expires, the information is sent back to the automated trading system, which records the result of the transaction in a "transaction history" database (internal logic file).

At the end of each trading day, a report (external output) is run that summarises the transactions from that day, and displays various calculations, such as the profit/loss made that day, number of trades of each commodity, average market price of all traded commodities, and an account summary. At any time, the user can request (external query) a detailed transaction history from between two dates, which lists the transaction ID, the commodity type, the bid/ask, and the price obtained (if any).

Step 1 — Categorising functions. The function categorisation has already been outlined above in the system description. From this description, we can see that there are two ILFs, and one of each of the remaining categories.

Step 2 — Assigning complexity values. For this, we consider only two items: the "price history" ILF, and the "daily report" EO.

For the ILF, the data consists of the commodity, and a history of low, high, open, and close prices on particular dates. Therefore, we identify one DET: the commodity name. We also identify one RET: the list of date/data pairs. For the purpose of this example, let's assume that the other ILF has the same configuration: one DET and one RET. We total two of each, so looking this up in Table 7.1, we get "simple".

For the EO, the report must access the trade information specific to the trading system, general market information, and account information. Therefore, it transacts with both ILFs and the EIF, making three FTRs. We identify three DETs: the transaction IDs on the report, the profit/loss, and the commodities. Looking this up in Table 7.1, we get "simple" again.

Step 3 — Calculating the count total. For the purpose of the example, we assume the rest of the ratings are "simple" as well. Filling out the template as in Table 7.3, we get the count total 29.

Information Domain Value	Count		Weighting Factor Simple	Average	Complex		
Internal Logical Files (ILFs)	2	×	**7**	~~10~~	~~15~~	=	14
External Interface Files (EIFs)	1	×	**5**	~~7~~	~~10~~	=	5
External Inputs (EIs)	1	×	**3**	~~4~~	~~6~~	=	3
External Outputs (EOs)	1	×	**4**	~~5~~	~~7~~	=	4
External Inquiries (EQs)	1	×	**3**	~~4~~	~~6~~	=	3
Count total							29

Table 7.3: A complete count total table for the automated trading system example.

Step 4 — Calculating the value adjustment factor. We will go over answers for only three questions; one at each end of the scale, and one around the middle.

1. Does the system require reliable backup and recovery? Rating: 3. The transaction history must be backed up nightly.

10. Is the internal processing complex? Rating: 5. Trend analysis is highly complex.

13. Is the system to be designed for multiple installations in different organisations? Rating: 0. The system will be installed and used on one machine within one organisation.

For the example, assume the rating for all other features is 2. This gives us the summation: $5+3+0+(11\times 2) = 30$.

Step 5 — Calculating the function point count The final step is a straightforward application of the formula:

$$\begin{aligned} \text{FP} &= \text{count total} \times (0.65 + 0.01 \times \textstyle\sum_{i=1}^{14} F_i) \\ &= 29 \times (0.65 + 0.01 \times 30) \\ &= 29 \times 0.95 \\ &= 27.55 \end{aligned}$$

Rounding up, we calculate 28 function points for the automated trading system.

7.3 Cost and Effort Estimation Models

One of the biggest challenges faced by the software engineering community is to achieve accurate estimates of the effort and cost taken to develop a system. Different methods range from arbitrary guesses, to machine learning techniques, to parametric models based on historical data.

Recent evidence suggests that parametric models provide the most accurate forecasts. In this section, we present the basic methods of cost estimation, and also discuss in detail the Constructive Cost Model II (COCOMO II), which is a hierarchical model for cost estimation that uses parametric methods.

7.3.1 Cost and effort estimation in software projects

The science of software estimation will never be exact, and probably will not even be considered at all accurate. A factor that makes this the case is simply that no person can reasonably predict what will go wrong in a project. Estimation methods and models typically assume that things will proceed as expected, and while we can simply add in some slack to take into account that things may go wrong, estimating this slack is difficult in itself.

This offers an extreme problem, because an inaccurate cost estimate can have many ramifications. If it is too high, it can cause a project not to go ahead at all. If it is too low, it can cost the developers much more than anticipated, perhaps resulting in a financial loss.

Pressman [Pre09] offers four solutions to the problem:

1. Delay estimation until one has as much information as possible. Preferably, we can wait until the end of the project, at which point we can offer an estimate that is 100% accurate!

2. Base estimates on data from previous projects that have been completed; especially projects that are solving similar problems.

3. Analyse the system, decomposing it into smaller parts, and generate estimates for these parts, rather than trying to estimate the entire project as one whole.

4. Use empirically-based estimation models.

The first option presents difficulties. Customers and managers want estimates as early as possible to decide whether the project should go ahead, and so they can start planning, thus delaying estimates may not be possible. Clearly, delaying until the end of the project is worthless. However, a premature estimate is just as worthless if it is not accurate, which it is unlikely to be. Another factor to consider is that, like other planning activities, estimates should be revisited as the project progresses. Each time estimation is revisited, we will have more information on which to base the estimate, which will (hopefully!) provide more accuracy.

The four solutions should not be considered in isolation. These solutions can work in tandem. Several cost estimation models insist on sufficient information to provide an estimate (point 1), are based on empirical evidence from previous projects (points 2 and 4), and provide methods to break the system down (point 3), as seen in the method for function point analysis in Section 7.2.3 (several models use function points as a basis for effort estimation).

There are many costs associated with software projects, such as the staff, facilities, software, and hardware. But for most projects, the biggest cost is the staff; specifically, the *effort* involved in running, implementing, and completing the project. It also seems to be the most difficult to estimate. For the remainder of this chapter, we will focus on effort estimation in software projects.

7.3.2 Expert Guesstimation

Effort in many projects is estimated using an expert's best judgement. Given a wealth of experience with similar projects, one may be able to provide an accurate estimate of effort. For example, given a project that seems to be of similar size and functionality as a previous project, we may use the final cost of the previous project as the estimate for the new project. If the expert judges that the new project is approximately half the size of the previous project, then the cost estimate may be roughly half.

Some expert-based methods involve polling several experts independently. For example, one technique involves asking several experts for three estimates: a pessimistic estimate (p), and optimistic estimate (o), and a most likely estimate (m). Each experts estimate is then calculated as

$$e = (p + 4m + o)/6.$$

Therefore, the most likely estimate is weighed four times as much as either the pessimistic or optimistic method. The overall estimate is then the average of all experts estimates.

The Delphi technique [SG05] asks several experts to make an individual judgement of the effort using any method they wish. Then, the average effort is calculated, and presented to all of the experts. Each expert is then given a chance to revise their estimate, in some cases after a discussion between all experts. This continues until no expert wishes to revise their estimate.

7.3.3 Parametric estimation

Based on empirical data, several researchers have present estimation models that relate the effort of project to several factors that influence the effort, such as the size of the project, the experience of the development team, and the type of system being developed.

Typically, these models are based on the theory that the size of the project is the most influential factor. The relationship between the factors and the effort is expressed as an equation of the form

$$E = a + bS^c m(\overrightarrow{X}), \tag{7.1}$$

in which S is the estimate size of the system (for example, in lines of code or function points), and b and c are coefficients (constants). The term $\overrightarrow{X}$ is a vector of the remaining cost factors, such as experience, and m is the adjustment multiplier for these factors. Informally, the above equation states that effort, E, is based mostly on the size of the project, but that an effort based on size must be adjusted by several other influencing factors about the project.

Several researchers and practitioners have applied regression analysis on data from over large groups of projects to calculate values for the coefficients in Equation 7.1. Many of these are LOC-based methods and function-point-based methods. Some of the most well-known LOC-based models proposed in the literature include:

E	$=$	$5.2\ \text{KLOC}^{0.91}$	Walston-Felix model
E	$=$	$5.5\ \text{KLOC}^{1.16}$	Bailey-Basili model
E	$=$	$3.2\ \text{KLOC}^{1.05}$	COCOMO-81 model
E	$=$	$5.288\ \text{KLOC}^{1.047}$	Doty model (for KLOC > 9)

In the above, KLOC stands for *thousands of lines of code*, and E is effort in person-months. Some of the most well-known function-point-based models include:

E	$=$	$-91.4 + 0.355$ FP	Albrecht and Gaffney model
E	$=$	$-37 + 0.96$ FP	Kemerer model
E	$=$	$-12.88 + 0.405$ FP	Small project regression model

There are a few interesting point of note here. First, in the above, none of the models contain adjustment multipliers, although in some cases, the authors of these models have included these into the model, but we omit them here.

Second, for each of the LOC-based models, the value of the constant coefficient a (from Equation 7.1) is always zero. Third, in the function-point models, the value of c is always one, which suggests that each function point adds only a linear amount of extra complexity to a project, whereas the LOC-based methods are polynomial. This is not the case for all function-point models, as we see in the next section.

An important lesson to take from the above example is about the level of maturity in the field of software engineering. If we estimate that we are required to implement 20KLOC, then we have the following estimates based on the four models discussed above:

$5.2 \times 20^{0.91}$	$=$	79.42	Walston-Felix model
$5.5 \times 20^{1.16}$	$=$	177.65	Bailey-Basili model
$3.2 \times 20^{1.05}$	$=$	74.34	COCOMO-81 model
$5.288 \times 20^{1.047}$	$=$	121.75	Doty model (for KLOC > 9).

So, from only four effort estimation models, we get a range from 75 person months to 178 person months. It is little wonder that software professionals find it difficult to estimate cost and effort of projects. A further point is that if we use the value of 28 for the function-based-based models, which is the number of functions points estimated in Example 7.2, the estimated number of person-months for all three models is negative!

However, one point does need to be made: the researchers who devised these models often recommend that individual organisations should calibrate the parameters for their own needs, and not rely on data from projects that are in different organisations, with different processes, that are likely for different domains.

7.4 COCOMO II for cost and effort estimation

COCOMO is a set of parametric cost and effort estimation models, first proposed by Boehm in 1981 [Boe81] (the original version is now called COCOMO-81) through various instantiations, and finally to COCOMO II [BAB$^+$00]. COCOMO II is in fact not a single model, but rather a hierarchy of empirical models based on project experience. The models are based on Boehm's analysis of a database of existing projects. Similar to Watson and Felix, Boehm's models are based on regression analysis of these systems. Since the formulation of COCOMO-81, there have been many changes in software engineering practices. COCOMO II is designed to accommodate different approaches to software development.

7.4.1 COCOMO II model hierarchy

COCOMO II is divided into three different models, with each one applied at a different stage of the development process:

1. **Application composition model** — used at the early stages of the process, before a requirements specification has been derived.
2. **Early design stage model** — used once requirements are relatively stable and the basic software architecture is available.
3. **Post-architectural stage model** — used during the detailed design, implementation, and test phases of the development lifecycle.

COCOMO requires the following size estimates to provide effort estimates, with each corresponding to the three different models above.

1. The number of *application points* for the application composition model.
2. The number of function points (Section 7.2.3 for the early design stage model.
3. The number of function points or source lines of code (Example 7.1) for the post-architectural stage model.

7.4.2 COCOMO II application composition model

The application composition model is applied early in the development lifecycle, before the requirements have been specified in detail. To estimate the size, the basic composition of the system being developed must be known, and this composition is used to estimate the effort in person-months.

The model describes a three step process:

1. Identify the *basic applications points* in the system. An application point is a measure similar to a function point, in that it attempts to measure the size and complexity by the amount of functionality of system.
2. Classify the complexity of each application point as *simple*, *medium*, or *difficult*.
3. Calculate the total number of application points using these weights.
4. Estimate the productivity rate of the project based on the experience of the development team, and the maturity of the development environment.
5. Calculate the effort from the number of application points and productivity rate.

The process of calculating the total number of application points is similar to that of calculating the number of function points (although there is less information available), described in Section 7.2.3, so we do not go into detail.

Step 1 — Identify the basic applications points.

The number of application points in a program is a weighted estimate of the following factors:

- the number of separate screens that are displayed in the application;
- the number of reports that are produced by the system; and
- the number of program components that are likely to be developed for the application.

Step 2 — Classify the complexity of the application points.

As with function points, the categories of application points (screen, report, and component) are each classified as either *simple*, *medium*, or *difficult*. The technique for doing this is similar to the estimation of complexity values in function point analysis, and tables very similar to Table 7.1 are used. In the case of application points, the rows represent the number of views contained in a screen, and sections contained in a report, while the columns represent the number of data tables hosted on client and server machines (for both screens and reports).

COCOMO II specifies that components should always be given classified as *difficult*, so estimating its complexity is not necessary[1].

Step 3 — Calculate the total number of application points.

The total number of application points is calculated in a similar manner to that of the total number of function points. Multiply the count of application points by the weight of the type, and then sum over all of these to give the total application point count. Formally:

$$\sum_{i=1}^{3}(AT_i \times W_i), \tag{7.2}$$

in which AT_i is the count of application types for screen, reports, and components, and W_i is the complexity weighting of these.

Application type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
Component	-	-	10

Table 7.4: Complexity weights for application types in COCOMO II.

The weight of an application type is derived from its complexity in step 2, and is dependent on whether it is a screen, report, or component. Table 7.4 shows the weights for the different types of application types and their complexities.

The model also accounts for maintenance projects, or projects in which some existing application points will be reused from previous projects. If r percent of application points will be used, then the number of *new* application points is simply

$$NAP_{new} = NAP \times \frac{100 - r}{100}.$$

Step 4 — Estimate the productivity rate.

The productivity rate is the number of application points that the team can do per person-month. The estimate for this is dependent on the experience of the development team for the particular type of application, and the maturity and capability of the development environment. Category ratings are described in Table 7.5.

Category	Description	
	Developer	Environment
Very low	Less than 5 months of experience.	Edit, code, debug.
Low	More than 5 months, less than 9 months of experience.	Simple front-end, with back-end support, little integration.
Nominal	More than 9 months, less than 1 year of experience.	Basic lifecycle tools, moderately integrated.
High	More than 1 year, less than 2 years of experience.	Strong, mature lifecycle tools, moderately integrated.
Very high	More than 2 years of experience.	Strong, mature, pro-active lifecycle tools, well integrated with processes, methods, reuse.

Table 7.5: Category ratings for development team experience, and environment maturity and capability.

Once the ratings of productivity for the development team and environment have been produced, the productivity rate of each is calculated using the values in Table 7.6. From this, the productivity rate of the project is simply the average of the two values.

Developer's experience and capability	Very low	Low	Nominal	High	Very high
Environment maturity and capability	Very low	Low	Nominal	High	Very high
Productivity	4	7	13	25	50

Table 7.6: Table for estimating the productivity rate of a project.

For example, given a development team with experience "high", using a development environment with maturity

[1]The rationale behind this is difficult to trace, and an explanation of why a component cannot have a simple or medium complexity appears absent from any literature we have read.

"low", we look up these values from Table 7.6, and get 25 for development team experience, and 7 for environment maturity. Therefore, the productivity rate of the project is $\frac{25+7}{2} = 16.5$.

Step 5 — Calculate the total effort.

The final step is to calculate the total effort estimate of the project. If *NAP* is the number of application points, and *PROD* is the productivity rate, then the total effort is simply

$$E = \frac{NAP}{PROD}.$$

7.4.3 COCOMO II early design and post-architectural models

Both the early design and post-architectural models have the same premise. Both are parametric models based on Equation 7.1, with the only difference being the parameters, and the coefficients.

These models are more sophisticated (and hopefully accurate!) that the application composition model, because they consider more factors in producing the estimate.

The basic equation of both of these models is the same:

$$E = bS^c m(\overrightarrow{X}), \tag{7.3}$$

in which the initial size estimate, bS^c, is adjusted by $m(\overrightarrow{X})$ which represents seven other influencing factors, known as *cost drivers*. The value for *b* is 2.94, based on calibration by the COCOMO II authors, however, they recommend that each organisation calibrates their own value for *b* based on historical data. The value of *c* is calculated by determining five *scaling factors* of a project.

Step 1 — Estimating size. The variable *S*, representing size, is the most influential factor in the estimation. In the COCOMO II models, size is represented as logical source lines of code (SLOC). Detailed checklists exist to determine what types of lines to include in the count.

At the early design phase, when the requirements are stabilising, it is difficult to estimate the number of lines of code, because much of the design is yet to be one. However, function point analysis (Section 7.2.3) can be applied to the requirements models, and a function point count can be derived.

If the team know which language will be used to develop the system, then they can estimate the number of lines of code from the number of function points. This is highly dependent on the language involved, as some languages are significantly more verbose than others. Table 7.7 shows the results of several studies that measured the number of logical lines of code in a whole host of systems written in different languages, and divided this by the number of function points in those systems.

Language	Average	Median	Low	High
Ada	154	-	104	205
Assembler	209	203	91	320
C	148	107	22	704
C++	59	53	20	178
C#	58	59	51	66
FORTRAN	90	118	35	-
Java	55	53	9	214
Perl	57	57	45	60
SQL	31	30	13	80
Visual Basic	50	52	14	276

Table 7.7: Number of logical lines of code per function point for various programming languages.

For example, consider a project with 200 function points. We can estimate from Table 7.7 that, if the system was implemented in C, it would be roughly 29,600 lines of code, whereas if implemented in Java, it would be roughly 11,000 lines of code.

Step 2 — Estimating scale. The scaling variable, c, is calculated by the project team. It is not a constant like the other parametric models discussed in Section 7.3.3. The basic equation for calculating c is

$$c = 1.01 + 0.01 \sum_{i=0}^{5} W_i, \tag{7.4}$$

in which W_i is one of five *scaling factors*, each with a value in the range 0 to 5. Thus, the value of the variable c can range from 1.01 (all factors are rated 0) to 1.26 (all factors are rated 5). The five scaling factors are:

1. **Precedentedness** — The level of which such applications have been attempted in the organisation before, ranging from 0 (thoroughly familiar) to 5 (thoroughly unprecedented).

2. **Development flexibility** — The level of flexibility in development process, methods, and tools, ranging from 0 (general goals) to 5 (rigorous).

3. **Architecture completed and risks eliminated** — Oddly, these are combined as one factor, and indicate the combined percentage of the number of interfaces specified and percentage of high-level risks that have been eliminated. Both are on the following scale:

Rating	Value
5	20%
4	40%
3	60%
2	80%
1	90%
0	100%

4. **Team cohesion** — The level of which a team can interact with each other, influenced by factors such as physical distance, communication mechanisms, and degree to which they have worked together before. Ranging from 0 (seamless interactions) to 5 (very difficult interactions).

5. **Process maturity** — A ranking use the Software Engineering Institute's Capability Maturity Model (CMM) [Hum90], ranging from 0 (chaotic, following no processes) to 5 (actively measuring and optimising processes)[2].

Step 3 — Estimating cost driver influence. The final step is to estimate $m(\overrightarrow{X})$, the cost drivers that influence the effort in a project. These multipliers reflect the capability of the developers, the non-functional requirements, the familiarity with the development platform, etc. In COCOMO II, the cost drivers for the early-design model consist of of seven different factors. Each of these factors is rated on a six point scale, from *very low* to *extra high*.

1. **RCPX** — the expected complexity of the internal processes, and the level of reliability required for the system.

2. **RUSE** — the level of reusability required. Note, this is not what will be reused from another system in this system, but the level of reuse that this system is expected to offer another system. A high level of reusability requires more effort due to the level of testing and documentation required, and the care required to design components to be generic.

3. **PDIF** — the level of *platform difficulty*. This refers to the constraints placed on the system by the platform on which it runs, such as the amount of processor time and storage available, as well as factoring in how frequent the underlying platform (e.g. software products, hardware resources) will change.

[2]In fact, the CMM only uses a scale of 1-5, not 0-5, but the ratings are similar for COCOMO II.

4. **PREX** — the experience of the personnel on the project. Ranging from less than 2 months (very low) to more than 6 years (very high). Strangely, there is no *extra high* rating for this.

5. **PERS** — the capability of the personnel on the project. Ranging from the 15th percentile (very low) to the 90 percentile (very high). Again, there is no *extra high* rating for this.

6. **SCED** — the constraints placed upon the project schedule, rated as a percentage of the "stretch-out" of the schedule. Schedules that are highly compressed (not stretched-out), require more effort than the optimal to complete the project on time. A rating of very low is a schedule that is 75% the length of the nominal project (recall from the Putnam-Norden-Rayleigh curve in Figure 5.1 that 75% percent is the most a schedule can be squeezed). A rating of very high is a project schedule that is 160%+ of the nominal. Empirical evidence suggests that compressing the schedule, and therefore expending less effort, results in lower quality software.

7. **FCIL** — the team support facilities. Similar to the environment maturity from Table 7.5.

Once the level of influence of these factors has been estimated, we assign them an explicit value from the rating in Table 7.8. Again, these parameters are calibrated by Boehm and his associates using data from existing projects. An interesting point is that, for the schedule stretch out (SCED), a rating *above* the nominal amount — that is, a schedule with a relaxed timeline — implies no decrease in effort.

Cost	**Rating**					
Driver	**Very Low**	**Low**	**Nominal**	**High**	**Very High**	**Extra High**
RCPX	0.75	0.88	1.00	1.15	1.30	1.66
RUSE		0.91	1.00	1.14	1.29	1.49
PDIF		0.87	1.00	1.11	1.27	1.62
PREX	1.23	1.11	1.00	0.89	0.82	
PERS	1.37	1.16	1.00	0.87	0.75	
SCED	1.29	1.10	1.00	1.00	1.00	1.00
FCIL	1.24	1.11	1.00	0.89	0.79	0.78

Table 7.8: Cost driver ratings for the COCOMO II early-design phase model.

The total multiplier for this is the product of these factors

$$\begin{aligned} m(\overrightarrow{X}) &= \times_{i=1}^{7} X_i \\ &= \text{RCPX} \times \text{RUSE} \times \text{PDIF} \times \text{PREX} \times \text{PERS} \times \text{SCED} \times \text{FCIL} \end{aligned}$$

For the post-architectural model, there are a total of 17 cost drivers, two of which overlap with the above (RUSE and SCED). These additional drivers factor in additional information that is typically unknown at the early-design phase, but is more likely to be known later in the development lifecycle. We do not discuss the post-architectural cost drivers in these notes.

Step 4 — Calculate the effort. The final step is to calculate the effort by simply filling in the parameters of Equation 7.3.

Step 5 — Calculate the time and personnel. COCOMO II can also be used to estimate the nominal amount of time that will be required to complete the project. The development time is calculated using the following equation:

$$T = 2.5E^{(0.33+0.2\times(c-1.01))} \times \frac{\text{SCED(Percentage)}}{100}, \tag{7.5}$$

in which SCED(Percentage) is the percentage estimate of SCED from step 3. The value of T is an estimate of the nominal delivery time for a project, such as $\mathrm{T_d}$ in the Putnam-Norden-Rayleigh curve in Figure 5.1.

From this, it is straightforward to calculate the estimated number of people, N, required to complete the project:

$$N = \frac{E}{T} \tag{7.6}$$

That is, the number of people is the effort (in person months) divided by the number of months.

Example 7.3. *Applying COCOMO II to the automated trading system*

Recall the automated trading system from Example 7.2. Now, we will estimate the effort that would be required to build that system.

In that example, the function point count was estimated at 28, which provides us with the size estimate (step 1). For step 2, we note that trading systems are not a new phenomenon, and in fact, the team has developed several trading systems before. However, this system is using new trend analysis techniques, so we estimate a rating of 3 for precedentedness. The team has some lightweight in-house development methodologies they used (rating of 1 for development flexibility), and 80% of the architecture will come from previous versions of the system (rating of 2 for architecture/risk). The team has been working together in a shared office for many years (rating of 1 for team cohesion) and are ranked as level 3 in the CMM.

Plugging these into Equation 7.4, for calculating the exponent c, we get

$$\begin{aligned} c &= 1.01 + 0.01 \sum_{i=0}^{5} W_i, \\ &= 1.01 + 0.01 \times (3 + 1 + 2 + 1 + 3) \\ &= 1.01 + 0.10 \\ &= 1.11. \end{aligned}$$

For step 3, we must estimate the cost drivers.

- **RCPX** — the complexity of trend analysis is high. Rating: very high.
- **RUSE** — a minimal level of reusability is required. Rating: low.
- **PDIF** — platform difficulty is constrained moderately constrained by the external systems to which we must connect. Rating: nominal.
- **PREX** — the team is highly experienced in these systems, although not the new type of trend analysis. Rating: very high.
- **PERS** — the team is highly capable. Rating: very high.
- **SCED** — there are no schedule constraints, so we will proceed with the nominal schedule (100%). Rating: nominal.
- **FCIL** — the support facilities are average. Rating: nominal.

$$\begin{aligned} m(\overrightarrow{X}) &= \times_{i=1}^{7} X_i \\ &= 1.66 \times 0.91 \times 1.00 \times 0.82 \times 0.75 \times 1.00 \times 1.00 \\ &= 0.929 \end{aligned}$$

Finally, we fill in Equation 7.3, using the default value of 2.94 for b, and assuming that we are using a language with an average of 50 lines of code per function point.

$$\begin{aligned} E &= bS^c m(\overrightarrow{X}) \\ &= 2.94(50 \times 28)^{1.11} \times 0.929 \\ &= 2.94 \times 3.106(KLOC) \times 0.929 \\ &= 8.48 \end{aligned}$$

Therefore, we estimate that the project will take approximately 8.5 person months to complete.

To calculate the nominal delivery time, we use Equation 7.5:

$$\begin{aligned} T &= 2.5E^{(0.33+0.2\times(c-1.01))} \times \frac{\text{SCED(Percentage)}}{100}, \\ &= 2.5 \times 8.48^{(0.33+0.2\times(1.11-1.01))} \times \frac{100}{100} \\ &= 2.5 \times 8.48^{0.35} \\ &= 2.5 \times 8.48^{0.35} \\ &= 5.28 \end{aligned}$$

Our estimate of 5.28 months can be now used to estimate the required number of staff using Equation 7.6:

$$\begin{aligned} N &= \frac{E}{T} \\ &= \frac{8.48}{5.28} \\ &= 1.606 \end{aligned}$$

Thus, we estimate that the project should take just over 1.5 people to complete in just over 5 months.

7.5 Effort Estimation in Agile Development

Parametric models for estimation are useful because they measure project size as the complexity of what needs to be done, and abstract away from factors such as the underlying software development lifecycle, which would have minimal impact. However, in agile development, projects typically have a short duration — or at least, the tasks within them have short durations. Furthermore, due to the lack of planning in agile development, techniques such as function point analysis cannot be used to estimate an entire project duration, because only some requirements are specified in enough detail up front.

In agile projects, there are two broad categories of models for effort estimation: one that estimates the size/complexity of users stories by comparison with other user stories, and one that estimates the size and effort of tasks within a period of time (called a *timebox*). In this section, we focus only on the former.

7.5.1 Estimating user stories by comparison

For agile projects in which development is guided by user stories, there are two key concepts that are used to effort estimation:

1. **Story points** — a story point is a relative measure of the size of a user story (recall from Section 3.3 that the requirements of the system are documented using user stories).
2. **Velocity** – velocity is a measure of productivity of team, which is represented by the number of story points delivered in a specified time period.

The estimation process

To estimate the effort and time required to complete a project using agile methodologies, a team will generally use a process similar to the following.

1. Divide the system up into stories, as discussed in Section 3.3.
2. Estimate the number of story points for each story, basing the estimate on the number of story points from previous stories.
3. Use the team's velocity from previous stories to estimate the delivery time of the project.
4. During development of a story, measure the actual velocity taken by the team.
5. Using this velocity, re-estimate the time that it will take to delivery the product.

Estimating story points

Story points are similar to function points, in that they measure the size and complexity of the system to be developed, rather than measuring the effort in time. However, unlike function points, they are not units of measure — they are relative measures. That is, while function points estimate the functionality delivered to the user by counting certain functional properties, story points "just are". The size of a story point is only relative to the size of other story points — so if story A is twice as big as story B, then story A should have twice as many story points. Thus, the first ever estimate of story points would have been arbitrary.

This measure causes problems between teams/organisations. If two teams are asked to calculate the function points in a requirements document, a technique can be followed such that their calculations will be similar. However, if two teams are asked to calculate the story points of a story, one team could estimate four, while the other could estimate four hundred. Neither would be more accurate than the other, because both estimates are relative to previous stories. When two teams come together for a project, or when a new member joins a team/organisation, this can cause problems. Despite this, story points are successfully used within organisations to estimate effort on agile projects.

There are several guidelines that can be used for estimating story points:

- **Estimate by analogy** — there are no units for story points, so the relative measure must come from other projects. If story A is about the same size as story B, they should have the same number of story points. If story A is twice the size, it should have twice as many story points. Therefore, always base your measures on other stories.

- **Decompose a story** — much like any planning task, some analysis can be useful. By decomposing a story into the high-level tasks that are required to complete the story, we can find measures that we know about the tasks, and combine them to provide a total measure.

- **Use the right units** — some agile proponents advise that the relative units should not be too fine grained. For example, can we distinguish a 35 point story from a 36 point story? If not, make the measures smaller. Typically, points are kept within a range, and a pattern-based scale is used. For example, measures can only be 1, 2, 4, 8, or 12. An 11 point story would likely be indistinguishable from to a 12 point story, so having both make little sense.

- **The "Doer" does the estimation** — story point estimation should be estimated by the team that is going to develop the story. This promotes ownership of the task.

- **Use group-based estimations** — for a story that is to be implemented by a team, the whole team should provide estimates. Techniques such as the Delphi method discussed in Section 7.3.2 can be used to reach a consensus.

When agile development was first becoming popular, many organisations used *ideal days* to measure effort. An ideal day is one in which a team member has no interruptions, such as meetings or phone calls, and can work solidly on a project. Story size was measured in the number of ideal days it would take to complete the task. While this provides a unit of measure, rather than relative measure, several drawbacks existed with this method. First, this is still a subjective measure. Second, and most importantly, it is a time-based measure, and experience from practitioners of agile methodologies indicates that stakeholders confuse ideal days with actual days, so start taking these as time estimates, instead of effort estimates. One must remember that ideal days are simply an attempt to provide consistency of estimates between different teams.

Some teams use ideal days as the basis for story points. That is, one story point equals one ideal day. Additionally, many new teams, who do not have a history of stories on which to base their estimates, use ideal days initially. Most teams are careful to use the term "points" instead of "ideal days" in their written estimate to stakeholders, so as to not have stakeholders mistaking the number of ideal days as an estimate of calendar time.

Measuring velocity

The velocity of a team is measured as the number of story points completed over a time period:

$$V = \frac{SP}{T_i},$$

in which SP is the number of story points completed, and T_i is the time period over which they were completed. For example, if a team completes 12 story points in one week, the velocity is 12 story points per week. If they complete eight more the following week, the velocity is 10 story points per week.

There are two common methods for measuring the velocity of a project for the purpose of estimation.

- **Using historical data** — by looking at the velocity of the team (or a similar team) over previous projects, we can calculate the average velocity of that team, and use this to form our estimates.
- **Using data from previous iterations** — by waiting until after the first iteration of a project, we can calculate the velocity of the team over that iteration, and use this to estimate the velocity of the remainder of the project. The velocity would be updated after each iteration.

Using both of these is probably the best approach. Initial estimates (before the first iteration) cannot use the second option, therefore, using historical data to estimate velocity seems reasonable. However, the first iteration is more likely to give an accurate estimate of future velocity for a specific team working on a specific project.

Estimate delivery time

To calculate the delivery time for a project, we simply sum the number of story points for all stories, and divide by the velocity:

$$T = \frac{\sum_{i=1}^{n} SP_i}{V},$$

in which SP_i is the number of story points in the i-th story.

Example 7.4. *Story point based estimation for the automated trading system*

We again return to our automated trading system in Example 7.2. Assume that there are five user stories identified for this, each with the following number of story points.

User story	Story points
Story 1	4 points
Story 2	8 points
Story 3	16 points
Story 4	16 points
Story 5	8 points
Total	52 points

The team of five that is developing this system has an average velocity of six user stories per fortnight. Thus, the initial estimate for delivery is

$$\begin{aligned} T_a &= \frac{\sum_{i=1}^{n} SP_i}{V} \\ &= \frac{\text{52 points}}{\text{6 points/fortnight}} \\ &= 8.66^{*} \text{ fortnights} \\ &\approx \text{17 weeks.} \end{aligned}$$

The team then takes six weeks to complete the first two iterations (stories). The first two stories together consist of 12 story points, meaning that the project velocity is four points per fortnight. Therefore, the delivery time needs to be re-calculated. However, we must be careful not to include the the story points that have been implemented in the first two iterations. We re-calculate using the new information to obtain the new delivery date:

$$\begin{aligned} T_b &= \frac{\sum_{i=1}^{n} SP_i}{V} \\ &= \frac{\text{40 points}}{\text{4 points/fortnight}} \\ &= \text{10 fortnights} \\ &= \text{20 weeks.} \end{aligned}$$

This new estimate specifies a delivery of 20 weeks from the current time (end of week 6); thus, 26 weeks from the start of the project.

7.5.2 Planning poker

Planning poker [Gre02] is perhaps the most commonly used estimation method in agile development today. It is a modification of the Delphi method discussed in Section 7.3.2.

It is commonly thought that group planning is more accurate that individual planning, due to the fact that in group planning, a wider range of expertise and experience is drawn upon. However, simply sitting down in a room and getting the group to talk about user stories can have negative results. In the original paper on planning poker, the author presents the following scenario:

> The customer reads a story to the team. Two guys are involved in discussing the impact of the story on the system. Reluctantly, an estimate is tossed out on the table. They go back and forth for quite a while. Everyone else in the room is drifting off, definitely not engaged. The discussion oscillates from one potential solution to another, avoiding putting a number on the card. When the discussion finally ends, you discover that the estimate did not really change over all that discussion. You just wasted 20 minutes of valuable time. You have 25 more stories to estimate, you don't want to make a career of release planning.

There are two major problems in this scenario. First, only two people have actually participated in the estimate. Second, it took too long to produce an estimate. Planning poker aims to help with these all-too-common problems with group estimation.

The process of planning poker

There are several variations of planning poker, but the original game proposed by Grenning proceeds as follows:

1. Each player is given an identical set of cards, with each card representing one estimate (in story points), for example, a set of cards containing the Fibonacci sequence 1, 2, 3, 5, 8, 13, and 21.

2. A *moderator*, who is sometimes the customer, but never takes part in the game, reads the user story; offering clarification if necessary.

3. Each player privately decides how many story points the story is "worth".

4. After each player has decided, all players simultaneously show their estimate using a card. This synchronisation is intended to remove the problem of people's estimates being influenced by other team members.

5. If the estimates do not correspond to some pre-decided termination criteria — for example, all estimates must be within a certain range of the average — then the people who estimated the lowest and highest story points are asked to explain their decision. The group then spend a short amount of time (about 2-3 minutes) discussing this. After the discussion, return to step 3.

6. If the estimates do correspond to the criteria, take the average as the estimate.

In most cases, the team will converge to an estimate that achieves the termination criteria. Different termination criteria can be used. For example, one criteria is that nobody changed their estimate from the previous round. If nobody is willing to adjust their estimate, additional rounds will continue indefinitely.

Using this method for estimation is an improvement on the initial scenario. First, everyone on the team must play the game, which makes full use of the experience and expertise in the team. Second, the discussion is timeboxed to prevent the most vocal people from dominating for too long.

There are still a number of weaknesses with this approach, most notably, the problem of *anchoring*. Anchoring occurs when the moderator or the player openly discuss their estimates or their expectations. For example, after the story has been read out, one player may say: "This one will be easy to implement", or "This is clearly at most a 10". At this point, anyone estimating significantly above 10 is likely to lower their estimate, especially if the person that spoke is influential; e.g. a respected developer. As such, it is important that before estimates are provided, there is no subjective discussion of the story.

In the original planner poker game, estimates are revealed at the same time to prevent anchoring. However, after the first round, all players know each other's estimates, therefore re-introducing the anchoring problem. As a result, a number of variations of the game have been proposed:

- The moderator collects all private estimates and calculates the average, revealing only this to the players. Players can discuss why they believe the average is too low or high, but cannot specify how far away their estimate is.
- In one variation, the average is presented and no discussion is permitted between estimates. The idea is that the players simply re-think their estimates based on the group answer.

Planning poker has been used with success on many projects, but it is not a technique for estimating the size of large-scale applications, or even large user stories. Planning poker works particularly well under the following conditions:

- when the team consists of people from a wide variety of backgrounds and with a wide variety of skills and expertise;
- when the story is suitable for one to two iterations on an agile project;
- when the team that will implement the story are the people that provide the estimates. Not only do they have an incentive to be accurate, but they know their skills and expertise better than others; and
- when the team have worked together on projects before, and therefore know the strengths of their team members.

Planning poker can be used if any of these conditions do not hold, however, it is likely to be less effective.

References

[BAB$^+$00] B. Boehm, C. Abts, W. Brown, S. Chulani, B. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece. *Software Cost Estimation with Cocomo II.* Prentice Hall, 2000.

[Boe81] B. Boehm. *Software Engineering Economics.* Prentice Hall, 1981.

[CKM94] SR Chidamber, CF Kemerer, and C. MIT. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[Eji91] L. Ejiogu. *Software Engineering with Formal Metrics.* QED Publishing, 1991.

[FC99] W.A. Florac and A.D. Carleton. *Measuring the software process: statistical process control for software process improvement.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[Gra92] R. B. Grady. *Practical Software Metrics for Project Management and Process Improvement.* Prentice-Hall, Englewood Cliffs, 1992.

[Gre02] J. Grenning. Planning Poker or How to avoid analysis paralysis while release planning. Technical report, Renaissance Software Consulting, 2002.

[Hum90] W.S. Humphrey. *Managing the software process.* Addison Wesley, 1990.

[MB03] S. Misra and V. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *Proceedings of Computational Science and Its Applications*, number 2667 in LNCS, pages 964–964. Springer, 2003.

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[SG05] A. Stellman and J. Greene. *Applied Software Project Management.* O'Reilly Media, 2005.

[WMW96] A.H. Watson, T.J. McCabe, and D.R. Wallace. Structured testing: A testing methodology using the cyclomatic complexity metric. NIST Special Publication 500-235, 1996.